XSS Vulnerability Detection Using Model Inference Assisted Evolutionary Fuzzing*

Position Statement

Fabien Duchene, Roland Groz, Sanjay Rawat, Jean-Luc Richier
UJF-Grenoble 1/Grenoble-INP/UPMF-Grenoble2/CNRS
Laboratoire d'Informatique de Grenoble UMR 5217
Grenoble F-38402, France {**duchene,groz,rawat,richier**}**@imag.fr**

***Abstract*—We present an approach to detect web injection vulnerabilities by generating test inputs using a combination of model inference and evolutionary fuzzing. Model inference is used to obtain a knowledge about the application behavior. Based on this understanding, inputs are generated using genetic algorithm (GA). GA uses the learned formal model to automatically generate inputs with better fitness values towards triggering an instance of the given vulnerability.**

***Index Terms*—Black-Box Security Testing, Genetic Algorithm, Test Automation, Model Based Fuzzing, Model Inference**

I. Introduction

Web vulnerabilities such as Cross Site Scripting (XSS) and SQL injections have been among the most targeted ones for several years [1]. Given the complexity of modern web based applications, naive blackbox fuzzing approches may not be sufficient to detect deeply nested vulnerabilities [2]. Indeed one of the problems of traditional fuzz testing is that it focuses more on data than on state transition. In the realm of blackbox testing, observing the state transition and generating inputs that traverse those states are not straightforward, which makes fuzzing less effective.

To address aforementioned problems, we propose to build a model of the SUT by using model inference techniques and guiding the fuzzing process by Evolutionary Algorithm. EA have already been used for generating test cases [3]. Our proposal is a smart-fuzzing approach for exhibiting deeply embedded injection vulnerabilities.

II. Approach Overview

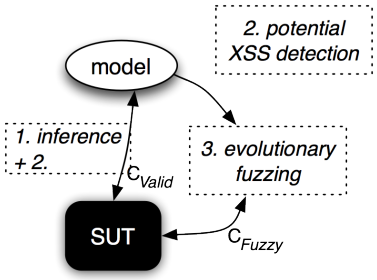

Fig. 1. Overview of our approach

*Work funded by project DIAMONDS (ITEA 2)

Currently, we only focus on detecting type-1 reflected XSS faults. Figure 1 shows the various components. SUT is a web application server. Model inference is responsible for learning the SUT state automaton. Based on the learned model, the evolutionary fuzzing component generates input sequences that are likely to exhibit XSS faults. During this second process, model inference can also enhance the SUT automaton.

A. Model Inference

Model inference is used to obtain a state model of the SUT. We use techniques developed in [4]. By learning a SUT formal w.r.t. some abstraction and concretization functions, we become aware of its state transitions and therefore can fuzz from some *appropriate* state.

B. Evolutionary Fuzzing

A candidate solution is a sequence of input parameters values. We use GA to generate malicious inputs sequences. The first generation is created using already accepted input sequences, the *attack grammar*, known attack inputs[5] - that worked for other SUT - and random sequences. Each individual is a sequence of input parameters values. Several populations evolve in parallel pools. Similar to [6], mutations and crossovers are performed both inside a given population and between different pools. Our fitness function is specific to XSS attacks and depends on the traversed states and the corresponding outputs. We use elitism for creating the next generation input.

We manually write an *attack grammar* G_i in order to guide both the mutation and crossover operators. It generates a subset of the inputs that attackers would attempt to submit to the SUT.

III. A deeper technical insight

A. Web Application

Let Σ be an alphabet. A transition u of a web application is a mapping from n user inputs $i_l^u \in \Sigma^*$: $I_u = i_1^u, ..., i_n^u$ to an output $q = q_1 \cdot q_2 \cdot ... \cdot q_k$, $q \in \Sigma^*$. Each q_j is either a webserver filtered input parameter i_l^u - i.e. $\exists f_r \in Filters, \underset{\sim}{q_j} = f_r(i_l^u)$ - or a string q_h surrounding one or two $\underset{\sim}{q_j}$. An individual is a sequence $I = (I_1, ..., I_m)$ where each I_u adheres to the above

definition. $Filters$ is a finite set of functions from $\Sigma^* \rightarrow \Sigma^*$ for example PHP addslashes(). A web application is modeled as an Input Output Labelled Transition System, in which each transition u conforms to the above definition.

B. Step 2 - Potential XSS detection

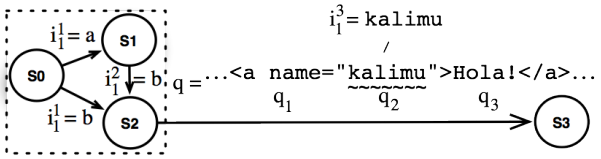

Fig. 2. When the value of an input parameter i_l is observed in the output q, the fuzzing starts from that initiating state on that very same i_l

The dash part contains some states that are accessible from the initial state S_0. The value of i_1^3 (`kalimu`) is observed in q, the output of the transition $S_2 \rightarrow S_3$. Thus we intuit that there is a possibility of reflected XSS on that transition. We then start fuzzing on i_1^3 from S_2.

C. An XSS Fuzzing Attack Grammar

We describe an input grammar to impose some restrictions on EA to generate inputs by constraining mutations and crossovers. This helps to be closer to attackers' behavior who would modify an interesting input parameter value to bypass filters f_r.

Let q be the following extract of the SUT output:

```
<a name="[USER_INFLUENCED_INPUT]">Hola!</a>
```

In this example $q = q_1 + q_2 + q_3$ meaning q_2 is the result of a filtering function applied to an input parameter i_l (i_1^3 in that case). Grammar fragment 1 shows an extract of the written attack grammar for guiding input mutations:

```
HTML_XSS_FIELD ::= HTML_TEXT_SIMPLE HTML_TAG_QUOTE
      HTML_TAG_SPACE HTML_TAG_EVENT HTML_TAG_EQUAL
      HTML_TAG_QUOTE JS_PAYLOAD
HTML_TAG_QUOTE ::= ' | "
HTML_TAG_SPACE ::= \n | \t | \r | ␣
HTML_TAG_EQUAL ::= =
HTML_TAG_EVENT ::= onabort | ... | onclick | ... | onwaiting
```

Grammar fragment 1. Injecting into an HTML attribute field value

Figure 3 shows an input parameter value i_l (thus a subset of an input sequence) generated using that grammar:

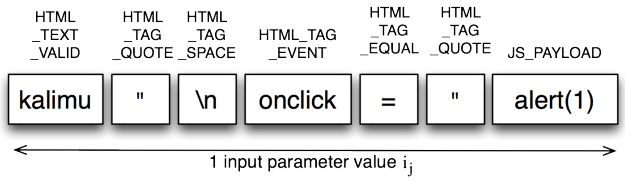

Fig. 3. An individual I is composed of several input parameters values $i_1^1 ... i_u^n$, each of them composed of at least one terminal in G_i

D. Creating the First Generation

Individuals of the first generation are created from the attack grammar and known attack inputs by reusing input sequences learned during the inferencing step that led to a state for which there is a potential injection on a given input parameter.

E. Character Classes

Exploiting an injection is about sending *data* and *instructions* to the SUT that does not use them in a safe way and assumes those inputs as only *data*. In our approach, we currently first submit only *data* for inferring a SUT formal model. Then during the fuzzing step, a combination of *data* and *instructions* is sent to the SUT. For now, the model is not updated during the fuzzing step. In the case of XSS, we consider the output grammar G_O to be HTML. Client browser parsers will render additional nodes in the parse tree of the output q depending of delimiting terminals, for instance C_1, C_2,... Thus input parameter values submitted during the inference step will only contain characters from $C_{valid} = C_0 \cup C_7$. We categorize symbols that appear in HTML words into the following classes:

- C_0: HTML Spaces: ␣|| \r || \t || \n
- C_1: HTML Attribute delimiter: " || ` || '
- C_2: HTML Tag delimiter: < || > || />
- C_3: HTML Equal sign: =
- C_4: JavaScript code: (||); ||{||}
- C_5: URL related: /|| : ||?||&
- C_6: Escaping character: \
- C_7: `HTML_TEXT_SIMPLE`: [a-Z] ∪ [0-9]

During the fuzzing step, input parameter values from $C_{fuzzy} = \cup_{i=0}^{7} C_i$ are submitted to the SUT.

F. Detecting XSS Attacks

The web application is vulnerable to XSS attacks if at least one output contains an attacker controlled JavaScript (JS) code. If the attacker succeeds in crafting an input i_l, s.t. $q_2 =$ **`kalimu" onclick="alert(1)`**, then q_2 is not syntactically confined - as defined in [7] - w.r.t. the SUT output grammar G_o (HTML5 in that example):

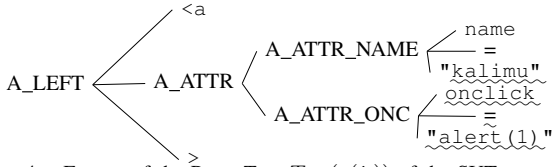

Fig. 4. Extract of the Parse Tree $T_{G_0}(q(i_l))$ of the SUT output q

Figure 4 shows an extract of the parse tree $T_{G_O}(q(i_l))$, where $T_G(q)$ is the parse tree a word q w.r.t. a grammar G.

This sufficient condition for detecting XSS attacks, however, has some false negatives [5] since modern browsers do accept HTML code that does not respect all G_0 production rules.

G. Evolutionary Fitness Function for XSS

The fitness function, denoted as $Fit(I)$, assesses how well a given individual I is close to detect an XSS, i.e. individuals with higher fitness value are prefered for creating next generation.

First, an interesting candidate reflects, in the output q_j, many character classes $C_{injected}(I)$ that were present in the

fuzzed input i_l for the very last transition: $C_{sent}(I)$. Thus $Fit(I)$ should be an increasing function of the number of classes present in $\underset{\sim}{q_j}$: $\frac{C_{injected}(I)}{C_{sent}(I)}$. In *Fig. 3.* and *Fig. 4.*, C_0, C_1, C_3, C_4, C_7 are present in i_l and are all successfully injected, thus $C_{injected}(I) = C_{sent}(I) = 5$.

States reachable within few transitions from the initial state are more likely to be sanitized than deeper ones, thus an injection within a transition further away from the start node is more likely to exist. We favor such individuals : $\frac{S_{reached}(I)}{S_{total}}$. In *Fig. 2.*, if $I = ((i_1^1 = a), (i_1^2 = b), (i_1^3 = \texttt{kalimu}))$, then $S_{reached}(I) = 3$

A well formed output q, w.r.t. G_O, is more likely to be executed by the client: $W_{ell}(I) = 1$ if the q is well formed, $W_{ell}(I) = 0$ otherwise.

Though an individual - $I = (I_1, ...I_m)$ for which a potential reflection does exist for i_l^m - is able to inject several character classes C_x, it is not enough to ensure that the reflections $\underset{\sim}{q(i_l^m)}$ will be interpreted as *instructions*. Additional HTML nodes in $T_{G_O}(q(i^m_{l_{fuzzed}}))$ compared to $T_{G_O}(q(i_l^m))$ indicates $i^m_{l_{fuzzed}}$ was able to successfully inject HTML *instructions*. We intuit that an attack has a higher probability to succeed in that case. Thus we make use of the proposed metric $N(I)$ that represents the improvement of I in terms of HTML nodes that are reflected from $i^m_{l_{fuzzed}}$ w.r.t. its predecessors $Pred(I)$. If $q(I_m) = q_1 \cdot q_2 \cdot ... \cdot q_k$, then $A(I) = \max_{j \in 1..k} Nodes_{G_O}(\underset{\sim}{q_j(i^m_{l_{fuzzed}})})$
For instance, in *Fig. 4.*, $A(I) = 3$.

$$N(I) = \frac{A(I) - \frac{\sum_{P \in Pred(I)} A(P)}{\|Pred(I)\|}}{\max_{E \in Gen} A(E)}$$

Finally, we propose the XSS fitness function:

$$Fit(I) = \frac{S(I)}{S_{total}} + \frac{C_{injected}(I)}{C_{sent}(I)} + W_{ell}(I) + N(I)$$

Each of these component can have weight to tune its impact.

H. Evolving the Population

Following we define mutation and crossover operations that tend to respect the Attack Input Grammar G_{AI}. Mutation operations would add, delete or replace at least one non-terminal or terminal w.r.t. G_{AI}.

Let $I = (I_1, ...I_m)$ and $J = (J_1, ..., J_w)$ being two individuals. A potential reflection does exist on I_m (resp. J_w), for the input parameter i_x^m (resp. j_y^w). Crossover is performed at two levels:

- transition level: a child would be $(I_1, ..., I_{m-1}, (i_1^m, ..., i_{x-1}^m, j_y^w))$
- input parameter value level: so far, two cases are considered: either i_x^m and j_y^w were generated using the same production rule or not. In the first case, we perform a classic 1 point crossover cutting on the same terminal production rule. It is still an ongoing reflexion on how to crossover input parameter values generated by different production rules while still conforming to the attack input grammar and wether there is an advantage of doing so instead of performing a 1 point crossover at a random position, as done in [8].

IV. Related work

Being a combination of two techniques viz. inference and genetic algorithm, our approach relates to several existing black-box testing works. In [8], the task of evolving malicious scripts is akin to generating malicious inputs also using an attack grammar. However, the absence of a SUT model (i.e. state transition to achieve the goal) might have some adverse effects, especially in the case of complex goals. The fitness function defined in [9], though being similar in goal, may not be effective in detecting deeply rooted vulnerabilities. KiF [10] uses model inference with manually crafted inputs for state transitions, whereas we tried to automatize this step using GA and the attack grammar. [11] is similar to our proposal: an abstract SUT model is inferred and concrete fuzzed input sequences are sent to the SUT. Differences include their use of passive inference and their only criteria for creating new input sequences is to increase the state coverage, probably because their targeted fault is SUT crash.

V. Conclusions and future work

We propose an automated type-1 XSS search approach that is based on model inference and evolutionary fuzzing to generate test cases. *Kameleon-Fuzz* is a work in progress implementation of our described approach. Our future work involves experimenting on real world applications, observing the influence of various GA parameters (elitism, pools, weights). We also plan to extend this approach for detecting type-2 XSS since current state of the art scanners detection capability is low [12]. Also, by considering the DOM and webserver as the SUT, it would be possible to detect type-0 XSS and non conformant XSS w.r.t. the HTML grammar. In that process, we will also tune our fitness function.

References

[1] CWE and SANS, "Top 25 most dangerous software errors."
[2] L. Butti and J. Tinns, "Discovering and exploiting 802.11 wireless driver vulnerabilities," Journal in Computer Virology, vol. 4, pp. 25–37, 2008.
[3] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," Information and Software Technology, vol. 51, pp. 957–976, 2009.
[4] "SPaCIoS Project no. 257876, FP7-ICT-2009-5," http://www.spacios.eu.
[5] M. Heiderich, "HTML5 security," http://html5sec.org/.
[6] J. D. DeMott, R. J. Enbody, and W. F. Punch, "Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing," 2007.
[7] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in Symposium on Principles of Programming Languages, 2006, pp. 372–382.
[8] J. Budynek, E. Bonabeau, and B. Shargel, "Evolving computer intrusion scripts for vulnerability assessment and log analysis," in Genetic and evolutionary computation, ser. GECCO '05, 2005.
[9] H. Srinivasan and K. Sarac, "A sip security testing framework," in Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference, ser. CCNC'09, 2009.
[10] H. J. Abdelnur, R. State, and O. Festor, "Kif: a stateful sip fuzzer," in Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications, ser. IPTComm '07, 2007.
[11] Y. Hsu, G. Shu, and D. Lee, "A model-based approach to security flaw detection of network protocol implementations," 2008.
[12] J. Bau, E. Bursztein, D. Gupta, and J. C. Mitchell, "State of the art: Automated blackbox web application vulnerability testing," in IEEE Symposium on Security and Privacy, 2010, pp. 332–345.